A Short Introduction to Scilab

Terence Leung, Tsing Nam Kiu

26 August 2006

1 About Scilab

Scilab is a freely distributed open source scientific software package, first developed by researchers from INRIA and ENPC, and is now developed by the Scilab Consortium. It is similar to Matlab, which is a commercial product. Yet it is almost as powerful as Matlab. Scilab consists of three main components:

- an interpreter,
- libraries of functions (Scilab procedures),
- libraries of Fortran and C routines.

Scilab is specialized in handling matrices (basic matrix manipulation, concatenation, transpose, inverse, etc.) and numerical computations. Also it has an open programming environment that allows users to create their own functions and libraries.

For more information and documentation, you may visit the Scilab homepage:

http://www.scilab.org

2 Installation and Execution of Scilab

First, you must have the software. To obtain one, go to the download section in the Scilab homepage. Find a right version for your operating system (platform) and then click to download. Please download the installer for binary version. Then double click the downloaded file and follow the instructions to complete the installation.

To execute Scilab, type **scilex** in the command prompt in the folder **bin** under the installation directory or click the shortcut in the start menu if you use Windows. Type **exit** or close the window of the main program to exit.

3 Documentation and Help

To find the usage of any function, type **help *function_name***, for example: **help sum**. If you want to find functions that you do not know, you can just type help and search for the keywords of the functions. Finally, if you want more information, you can visit the Scilab homepage. There is a section called documentation. It is very resourceful.

4 Scilab Basics

4.1 Common Operators

Here is a list of common operators in Scilab:

$+$	Addition
$-$	Subtraction
$*$	Multiplication
$/$	Division
$\hat{}$	Power
$'$	Complex conjugate transpose

4.2 Common Functions

Some common functions in Scilab are: sin, cos, tan, asin, acos, atan, abs, min, max, sqrt, sum. E.g., when we enter:

```
sin(0.5)
```

then it displays:

```
ans: =
     0.4794255
```

Another example:

```
max(2, 3, abs(-5), sin(1))
ans: =
      5.
```

4.3 Special Constants

We may wish to enter some special constants like, i (sqrt(-1)) and e. It is done by entering **%pi**, **%i** and **%e** respectively. There are also constants **%t** (true) and **%f** (false) which are Boolean variables. Boolean variables would be introduced later.

4.4 Data Structures

Scilab supports many data structures. Examples are: usual (real or complex matrices), polynomial, Boolean, string, function, list, tlist, sparse, library. Please read Scilab documentation for details. To query for the type of an object, type: **typeof(*object*)**.

4.5 Strings

To enter strings, enclose the string with either single or double quotations. For example: **‘ This is a string’** or **“this is also a string”**.

To concatenate strings, use the operator + :
“Welcome ” + “to ” + “Scilab!”
ans: =
Welcome to Scilab!

There are some basic string handling functions such as **strindex**, **strsplit**, **strsubst** and **part**. Please refer to Scilab's documentation for details.

4.6 Saving and Loading Variables

To save and load variables, we use save and load functions:
save(‘*file_name*’, *var1*, *var2*, . . .);
load(‘*file_name*’, ‘*var1*’, ‘*var2*’, . . .);
where ***file_name*** is the name of the file to be saved or loaded, and ***var1***, ***var2***, . . . are names of variable.

Notice that the variable name must match the name when it is to be saved. Here are some illustrations.

```
a = 3; b = %f; s = ‘scilab’;
save(‘save.dat’, a, b, s);
clear a; // delete the variable a
clear b;
clear s;
load(‘save.dat’, ‘a’, ‘b’, ‘s’);
// load all the saved variables

load(‘save.dat’,‘b’);
// It loads only variable b, but not
// variable a in the name of b

load(‘save.dat’,‘d’);
// It will not show any error messages.
// Variable d is undefined, not empty.

listvarinfile(“save.dat”);
// list variables in a file saved by
// the function save
```

Name	Type	Size	Bytes
a	constant	1 by 1	24
b	boolean	1 by 1	20
s	string	1 by 1	44

5 Dealing with Matrices

5.1 Entering Matrices

There are many ways to enter a matrix. Here is the simplest method:

1. separate each elements in a row using a blank space or a comma;
2. separate each row of elements with a semi-colon;
3. put the whole list of elements in a pair of square brackets.

For example, we wish to enter a 3×3 magic square and assign to the variable M.

```
M = [8 1 6; 3 5 7; 4 9 2]
M =
   8.   1.   6.
   3.   5.   7.
   4.   9.   2.
```

5.2 Calculating Sums

For a magic square, we wish to check for its column sums and row sums and the sum of diagonals. This is done by entering:

```
sum(M,‘c’)
ans: =
      15.
      15.
      15.

sum(M,‘r’)
ans: =
      15. 15. 15.
```

The sum of the main diagonal is easily done with the help of the function **diag**.

```
diag(M)
ans: =
      8.
      5.
      2.
```

5.3 Subscripts

It is a bit more difficult to find the sum of the other diagonal. We will show two ways to accomplish it. One method is to find the sum manually, i.e., to read the appropriate elements and then to sum them up.

```
M(1,3) + M(2,2) + M(3,1)
ans: =
      15.
```

It is possible to access elements in a matrix using a single index. This by treating a matrix as a long vector formed by stacking up the columns of the matrix. E.g.: **M(1) = 8**, **M(2) = 3**, **M(3) = 4**, **M(4) = 1**, **M(5) = 5**, ...

Accessing out-of-bound elements will result in an error, like entering:

```
M(3,4)
        !–error 21
invalid index
```

A smarter way to get the sum of the other diagonal is to use the function **mtlb_fliplr**, where **mtlb**stands for **Matlab**. It is to flip a matrix left-to-right (lr):

mtlb_fliplr(M)
ans =
6. 1. 8.
7. 5. 3.
2. 9. 4.

The desired result would be obtained by typing: **sum(diag(mtlb_fliplr(M)))**.

5.4 The Colon Operator

The colon operator is one of the most important operators in Scilab. The expression **1:10** results in a row operator with elements 1, 2, . . . , 10, i.e.

1:10
ans =
1. 2. 3. 4. 5. 6. 7. 8. 9. 10.

To have non-unit spacing we specify the increment:

10 : -2 : 2
ans =
10. 8. 6. 4. 2

Notice that expressions like **10:-2:1**, **10:-2:0.3** would produce the same result while **11:-2:2** would not.

Subscript expressions involving colons refer to parts of a matrix. **M(i:j, k)** shows the i-th row to j-th row of column k. Similarly,

M(3,2:3)
ans =
9. 2.

Some more examples:

M(3,[3,2])
ans =
2. 9.

M([2,1], 3:-1:1)
ans =
7. 5. 3.
6. 1. 8.

The operator $, which gives the largest value of an index, is handy for getting the last entry of a vector or matrix. For example, to access all elements except the last of the last column, we type:

M(1:$-1, $)

We sometimes want a whole row or a column. For example we want all the elements of the second row of M. We enter:

M(2,:)
ans =
3. 5. 7.

Now we have a new way to perform operations like **mtlb_fliplr(M)**. It is done by entering **M(:, $:-1:1)**. However the function **mtlb_fliplr(M)** would obtain result faster (in computation time) than using the subscript expression.

5.5 Simple Matrix Generation

Some basic matrices can be generated with a single command:

zeros	all zeros
ones	all ones
eye	identity matrix (having 1 in the main diagonal and 0 elsewhere)
rand	random elements (follows either normal or uniform distribution)

Some illustrations:

zeros(2,3)
ans =
0. 0. 0.
0. 0. 0.

8 * ones(2,2)
ans =
8. 8.
8. 8.

eye(2,3)
ans =
1. 0. 0.
0. 1. 0.

rand(1,3,'uniform') // same as rand(1,3)
ans =
0.2113249 0.7560439 0.0002211

5.6 Concatenation

Concatenation is the process of joining smaller size matrices to form bigger ones. This is done by putting matrices as elements in the bigger matrix:

a = [1 2 3]; b = [4 5 6]; c = [7 8 9];
d = [a b c]
d =
1. 2. 3. 4. 5. 6. 7. 8. 9.

e = [a; b; c]
e =
1. 2. 3.
4. 5. 6.
7. 8. 9.

Concatenation must be row/column consistent:

x = [1 2]; y = [1 2 3];
z = [x; y]
!–error 6
inconsistent row/column dimensions

We can also concatenate with block matrices, e.g.:

[eye(2,2) 5*ones(2,3); zeros(1,3) rand(1,2)]
ans =
1. 0. 5. 5. 5.
0. 1. 5. 5. 5.
0. 0. 0. 0.6525135 0.3076091

Remember that it is an error to access out-of-bound element of a matrix. However, it is okay to assign values to out-of-bound elements:

```
M = matrix(1:6, 2, 3); M(3,1) = 10
M =
   1.    3.    5.
   2.    4.    6.
   10.   0.    0.
```

It is remarked that this method is slow. If the size of the matrix is known beforehand, we should use pre-allocation:

```
M = zeros(3,3); // pre-allocation
M([1 2], :) = matrix(1:6, 2, 3);
M(3,1) = 10;
```

5.7 Deleting Rows and Columns

A pair of square brackets with nothing in between represents the empty matrix. This can be used to delete rows or columns of a matrix. To delete the 1st and the 3rd rows of a 4x4 identity matrix, we type:

```
A = eye(4,4);
A([1 3],:) = []
A =
   0.   1.   0.   0.
   0.   0.   0.   1.
```

If we delete a single element from a matrix, it results in an error, e.g.:

```
A(1,2) = []
         !–error 15
submatrix incorrectly defined
```

If we delete elements using single index expression, the result would be a column vector:

```
B=[1 2 3; 4 5 6];
B(1:2:5)=[]
B =
   4.
   5.
   6.
```

5.8 Matrix Inverse and Solving Linear Systems

The command **inv(M)** gives the inverse of a matrix M. If the matrix is badly scaled or nearly singular, a warning message will be displayed:

```
inv([1 2;2 4.0000001])
warning
matrix is close to singular or badly scaled.
rcond = 2.7778D-09
ans: =
         40000001.    - 20000000.
       - 20000000.      10000000.
inv([1 2;2 4])
         !–error 19
Problem is singular
```

Solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$, i.e., to find x that satisfies the equation, when **A** is a square, invertible matrix and **b** is a vector, is done in Scilab by entering **A \ b** :

```
A = rand(3,3), b = rand(3,1)
A =
   0.2113249   0.3303271   0.8497452
   0.7560439   0.6653811   0.6857310
   0.0002211   0.6283918   0.8782165
b =
   0.0683740
   0.5608486
   0.6623569
x = A \ b
x =
 - 0.3561912
   1.7908789
 - 0.5271342
```

Another method is to type **inv(A) * b**. Although it gives the same result, it is slower than **A \ b** because the first method mainly uses Gaussian Elimination which saves some computation effort. Please read the Scilab help file for more details about the slash operator when **A** is non-square.

A / b solves for **x** in the equation: **xb** = **A**.

5.9 More on Handling Matrices

To add 4 to each entry of a matrix M, using **M + 4 * ones(M)** is correct but troublesome. Indeed this can be done easily by **M + 4**. Subtraction of a scalar from a matrix entry-wise is done similarly.

Multiplying 2 to the second column and 3 to the third column of M can be achieved by using the entry-wise multiplication operator **.*** : **M .* [1:3; 1:3]**.

Entry-wise arithmetic operations for arrays are:

+	Addition
−	Subtraction
.*	Multiplication
.^	Power
./	Right division
.\	Left division

To enter the matrix **M = [1 2 3 4 5; 6 7 8 9 10]**, one may use:

```
M = zeros(2,5);
M(:) = 1:10
```

Yet an almost effortless method is to use the function matrix, which reshapes a matrix to a desired size.

```
M = matrix(1:10,2,5)
```

How to enter **N = [1 2; 3 4; 5 6; 7 8; 9 10]** easily? Hint: think of some simple operations on a matrix.

A handy function in Scilab called **size** returns the dimensions of the matrix in query:

```
size(M)
ans: =
       2.  5.
```

while **size(M,1)** and **size(M,2)** return 2 (number of rows) and 5 (number of columns) respectively.

6 More About Command Line

Entering a semi-colon at the end of a command line suppresses showing the result (the answer of the expression). One example was from the section on Concatenation:

a = [1 2 3]; b = [4 5 6]; c = [7 8 9];

It suppresses the showing of variables a, b and c.

A long command instruction can be broken with line-wraps by using the ellipsis (. . .) at the end of each line to indicate that the command actually continues on the next line:

s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
- 1/8 + 1/9 - 1/10 + 1/11 - 1/12;

Are there any ways to calculate the above expression more easily? A solution is:

s = sum((1:2:12) .\ 1) - sum((2:2:12) .\ 1)
s = sum(1 ./ (1:2:12)) - sum(1 ./ (2:2:12))

Note that **1./1:2:12** is interpreted as **(1.)/(1:2:12)**. Similarly, **1:2:12.\1** is interpreted as **1:2:(12.)\1**.

Using the up and down arrow in the command line can recall previous commands.

7 The Programming Environment

7.1 Creating Functions

Scilab has an open programming environment that enables users to make their own functions and libraries. It is done by using the built-in editor called SciPad. To call the editor, type **scipad()** or **editor()**, or click Editor at the menu bar.

The file extensions used by scilab are **sce** and **sci**. To save a file, click for the menu File and choose Save. To load a file, choose Load under the same menu. To execute a file, type **exec('*function_file_name*')**; in the command line or click for load into Scilab under the menu Execute.

To begin writing a function, we type:

function[*out1 out2*,. . .]=**name**(*in1 in2*,. . .)

where function is a keyword that indicates the start of a function, ***out1, out2***,. . . and ***in1, in2***,. . . are variables that are output and input of the function respectively, the variables can be Boolean, numbers, matrices, etc, and name is the name of the function. Then we can enter the body of the function. At the end, type

endfunction

to indicate the end of the function. Comment lines begin with //. A sample function is given as below:

```
function [d] = distance(x, y)
// this function computes the distance
// between the origin and the point (x, y)

d = sqrt(x^2 + y^2);

endfunction
```

Unlike Matlab, Scilab allows multiple function declaration (with different function names) within a single file. Also Scilab allows overloading (it is not recommended for beginners). Please refer to the chapter overloading in its help file for details.

7.2 Flow Control

A table of logical expressions is given below:

==	equal
∼=	not equal
>=	greater than or equal to
<=	less than or equal to
>	greater than
<	less than
∼	not

If a logical expression is true, it returns a Boolean variable **T** (true), otherwise **F** (false).

The **if** statement: It has the basic structure:

```
if condition
  body
end
```

The body will be executed only when the condition statement is true. Nested **if** statements have the structure:

```
if condition1
  body1
elseif condition2
  body2
elseif conditon3
  body3
elseif ...
end
```

For example,

```
s = input('please input a number')
// ask for a number and store in s
if s > 0
   disp('It is positive.');
elseif s < 0
   disp('It is negative.');
else
   disp('It is not a positive...
   or negative number.');
end
```

The **for** loop: It has the basic structure:

```
for variable = i : step : j
body
end
```

The loop will be executed a fixed number of times specified by the number of elements in the array variable. A slightly modified version is:

```
str = 'abcdr';
s = ''; // an empty string
for i = [1 2 1 3 1 4 1 2 5 1]
  s = s + part(str, i);
end
disp(s); // s = abacadabra
```

The **while** loop: It has the basic structure:

```
while condition
 body
end
```

The loop will go on as long as the condition statement is true. Here we give an example of the Euclidean Algorithm.

```
function [n1] = hcf(n1, n2)
// n1 and n2 are positive integers

if n2 > n1
 tem = n2; n2 = n1; n1 = tem;
 // to ensure n1>=n2
end

r = pmodulo(n1, n2);
 // remainder when n2 divides n1
n1 = n2; n2 = r;

while r ~= 0
 r = pmodulo(n1, n2);
 n1 = n2; n2 = r;
end

endfunction
```

The **break** and the **continue** commands: To end a loop and to immediately start the next iteration, respectively. For example:

```
// user has to input 10 numbers and for
// those which are integers are summed up,
// the program ends prematurely once a
// negative number is entered
result = 0;
for i = 1:10
 tem = input('please input a number');
 if tem < 0
  break;
 end
 if tem ~= int(tem) //integral part
  continue;
 end
 result = result + tem;
end
disp(result);
// It is not well written, just to
// illustrate the use of the two commands
```

7.3 Some Programming Tips

The concept of Boolean vectors and matrices is important. The function find is useful too. It reports the indices of true Boolean vectors or matrices. For example:

```
M = [-1 2; 4 9]; M > 0
ans: =
     F   T
     T   T
M(M>0)'
ans: =
     4. 2. 9.
```

In contrast,

```
find(M>0)
ans: =
     2. 3. 4.
M(find(M>0))'
ans: =
     4. 2. 9.
```

We remark that **M(M>0)** is quicker than **M(find(M>0))** because the **find** function is unnecessary in this case.

It is important to distinguish **&** and **and**, | and **or**. The first one of each pair is entry-wise operation and the other one reports truth value based on all entries of a Boolean matrix.

```
M = [0 -2; 1 0]; M==0 | M == 1
ans: =
     T   F
     T   T

and(M>=0) // true iff all entries are true
ans: =
     F

or(M == -2) // false iff all entries are false
ans: =
     T
```

7.4 Debugging

The most tedious work in programming is to debug. It can be done in two ways: using Scilab's built-in debugger, or modifying the program so that it serves the same purpose as the debugger. The debugger is similar to those debuggers in other programming languages and is simple to use. We present the second method to offer programmers greater flexibilities when debugging.

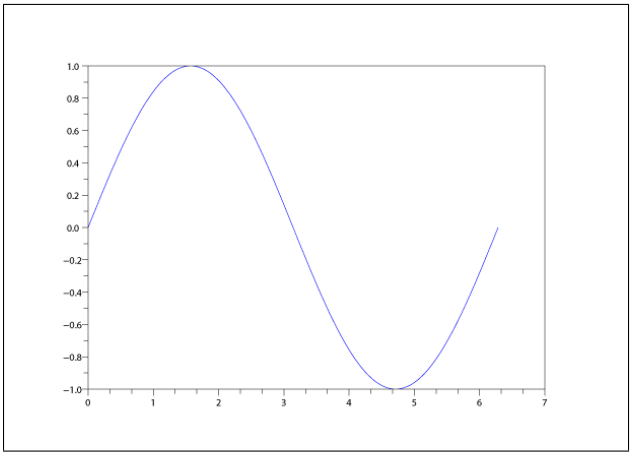
To insert breakpoints we use **pause**. To end **pause** we use **abort**. To set the output of the function we may use **return**. To display variables we use **disp(*variable_name*)**. For details please read the Scilab documentation.

8 Plotting Graphs

8.1 Plotting 2D Graphs

The plot function has different forms, depending on the input arguments. If y is a vector, **plot(y)** produces a piecewise linear graph of the elements of y versus the index of the elements of y. If you specify two vectors as arguments, **plot(x,y)** produces a graph of y versus x. To plot the value of the sine function from zero to 2:

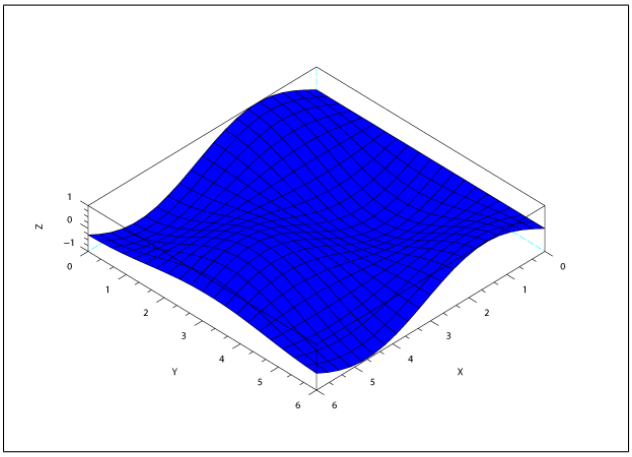
```
t = (0:1/100:2) * %pi;
y = sin(t);
plot(t,y);
```



8.2 Plotting 3D Surfaces

The command **plot3d(x,y,z)** plots 3D surfaces. Here x and y (x-axis and y-axis coordinates) are row vectors of sizes n1 and n2 and the coordinates must be monotone, and z is a matrix of size (n1,n2) with z(i,j) being the value of the surface at the point (x(i),y(j)).

```
// simple plot using z=f(x,y)
t=[0:0.3:2*%pi]';
z=sin(t)*cos(t');
plot3d(t,t,z)
```



9 Scilab versus Matlab

This section is based on some user comments found in the internet, thus not necessarily all true. It is intended to give readers a general image about their differences besides those in syntax.

- Matlab has a thorough documentation; the one in Scilab is brief.
- Matlab has a lot of optimization on computation, thus it is faster than Scilab.
- Matlab has a very powerful simulation component called Simulink. Scilab has Scicos that serves the same purpose but it is weaker.
- Matlab has a much better integration with other programming languages and programs such as C, C++ and Excel.
- The graphics component of Scilab is weak (has fewer functions).
- Most importantly, Scilab is **FREE**. It certainly outweighs its deficiencies. It is remarked that Scilab is more than enough for casual and educational uses.

References

[1] Scilab help file (its own documentation)

[2] Scilab for dummies:
http://www–irma.u–strasbg.fr/~sonnen/SCILAB_HELP/scilab_for_dummies.htm

[3] Matlab primer:
http://ise.stanford.edu/Matlab/matlab–primer.pdf